Écrire des logs en Python

67

This entry was posted in Programmation *on 08/03/2013 by Sam*

Good morning…

Au lieu de mettre des `print()` partout qu'il va falloir retirer après et qui en plus ne servent à rien dans un processus daemonisé, utiliser les facilités de logging de Python peut se révéler un bon investissement. Investissement car le module `logging` est du même genre que `urllib2`, `datetime` ou `os.path` : on peut tout faire avec mais vaut mieux avoir la doc sous la main.

Pour les gens pressés

Avant de se lancer dans les explications, voici le snippet qui permet d'afficher les informations à l'écran et dans un fichier de log :

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import logging

from logging.handlers import RotatingFileHandler

# création de l'objet logger qui va nous servir à écrire dans les logs
logger = logging.getLogger()
# on met le niveau du logger à DEBUG, comme ça il écrit tout
logger.setLevel(logging.DEBUG)

# création d'un formateur qui va ajouter le temps, le niveau
# de chaque message quand on écrira un message dans le log
formatter = logging.Formatter('%(asctime)s :: %(levelname)s :: %(message)s'
# création d'un handler qui va rediriger une écriture du log vers
# un fichier en mode 'append', avec 1 backup et une taille max de 1Mo
file_handler = RotatingFileHandler('activity.log', 'a', 1000000, 1)
# on lui met le niveau sur DEBUG, on lui dit qu'il doit utiliser le formate
# créé précédement et on ajoute ce handler au logger
file_handler.setLevel(logging.DEBUG)
file_handler.setFormatter(formatter)
logger.addHandler(file_handler)
```

```
# création d'un second handler qui va rediriger chaque écriture de log
# sur la console
steam_handler = logging.StreamHandler()
steam_handler.setLevel(logging.DEBUG)
logger.addHandler(steam_handler)

# Après 3 heures, on peut enfin logguer
# Il est temps de spammer votre code avec des logs partout :
logger.info('Hello')
logger.warning('Testing %s', 'foo')
```

Cette config va afficher :

```
Hello
Testing foo
```

sur la console

Et :

```
2013-03-08 11:37:31,311 :: INFO :: Hello
2013-03-08 11:37:31,411 :: WARNING :: Testing foo
```

Dans le fichier de *activity.log*.

Quand vous passez en prod, mettez simplement une condition sur le stream handler, et le `setLevel` sur `logging.WARN` et vous êtes tranquille, quel que soit le nombre de `logger.info` que vous avez mis partout.

Mais bon, avouez le, si vous aviez dû le faire vous même, vous auriez abandonné avant de trouver. D'une manière générale, quand vous voyez des noms en camelCase comme `setLevel` dans une lib Python, c'est que la lib va être relou à utiliser.

Donc, faites comme moi, mettez cette config dans un fichier à part, et pour chaque nouveau projet, copier le bêtement. Comme ça vous aurez en une seconde la possibilité de logger dans un fichier et sur la console avec une désactivation facile.

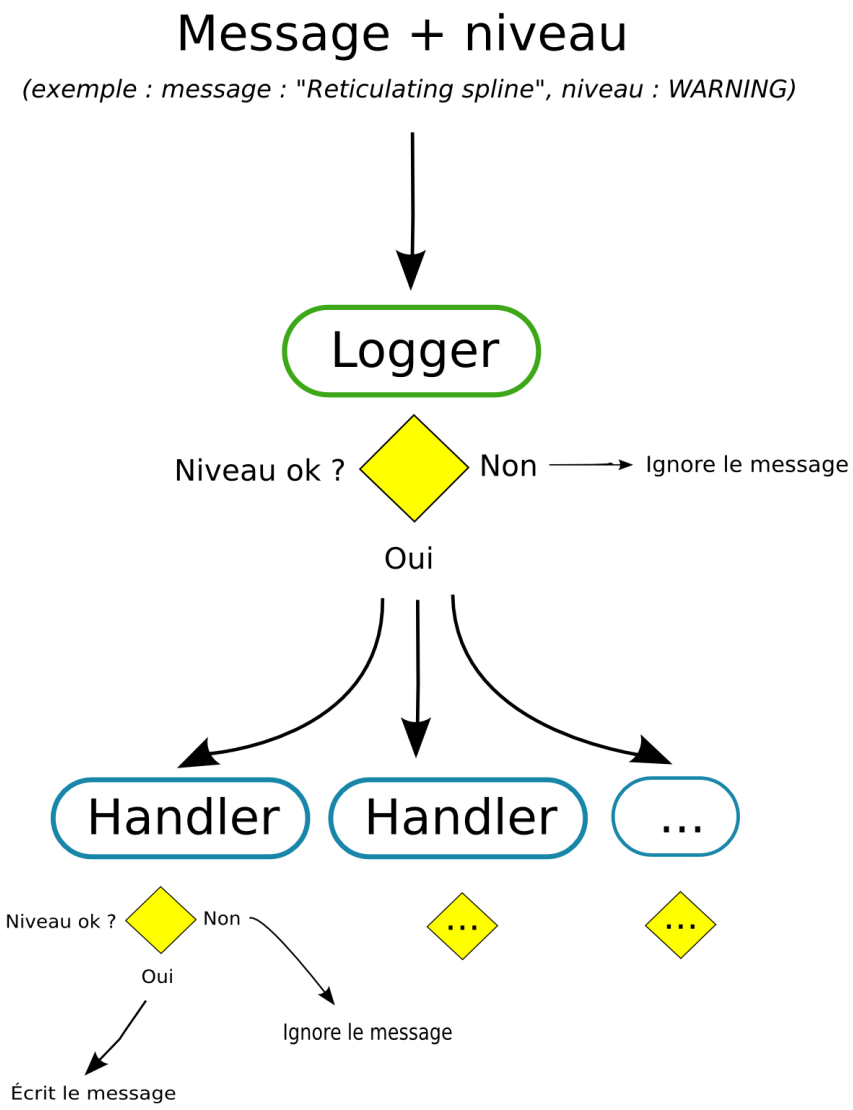
Un peu de théorie

En Python, les logs se font à travers un logger. Le logger prend toute écriture que vous lui demandez, et regarde si le niveau de log est suffisamment haut pour continuer. Si non, il ignore l'entrée, si oui, il va passer le message à chaque handler.

Chaque handler fait le même test pour lui même : le niveau est il assez élevé pour moi ? Si non, il ne fait rien. Si oui, il écrit le log. La manière dont il écrit le log dépend du type du logger : le StreamHandler va écrire par défaut sur la console, le FileHandler dans un fichier, le SmtpHandler va envoyer un email, etc. Un logger peut avoir autant de handlers qu'il veut, et donc autant de type d'écritures qu'il veut.

Tout handler possède un formatter, qui est un objet qui décide dans quel format il va écrire la nouvelle entrée du log. Le formateur par défaut écrit juste le message, mais on peut lui demander (comme on l'a fait plus haut), d'écrire un timestamp, le niveau du message ou des choses plus complexes comme des éléments de la pile d'appel.

Vous avez suivi ? Allez, comme je suis sympa, je vous mets un dessin :



Si vous saviez combien de temps prennent ces schémas à la con à faire...

Bref, pour configurer un logger en Python :

- On créé un objet logger
- On lui ajoute des handlers
- On ajoute des formateurs aux handlers
- On set les niveaux du loggers et des handlers

Et enfin, on peut logger avec `logger.(info|warn|debug|error|critical)`.

Mais là vient s'intercaler un des plus grands mystères de l'histoire du logging : les niveaux. Personne ne se rappelle comme ça marche.

Si vous mettez un flingue sur la tête de quelqu'un et que vous lui donnez une dernière chance d'expliquer les niveaux de logging sinon vous le butez lui et toute sa famille, il se mettra à chouiner bêtement. Inutile de monter la pression en menaçant de profaner leurs tombes et violer leurs cadavres. Il ne sait tout simplement pas.

Donc, un bonne fois pour toute, voilà comment les niveaux marchent

Niveau	Valeur	Usage

CRITICAL	50	Le programme complet est en train de partir en couille.
ERROR	40	Une opération a foirée.
WARNING	30	Pour avertir que quelque chose mérite l'attention : enclenchement d'un mode particulier, detection d'une situation rare, un lib optionelle peut être installée.
INFO	20	Pour informer de la marche du programme. Par exemple : "Starting CSV parsing"
DEBUG	10	Pour dumper des information quand vous débuggez. Par exemple savoir ce qu'il y a dans ce putain de dictionnaire.

Vous spécifiez les niveaux une fois quand vous créez vos loggers et handlers avec `setLevel`. Vous passez à cette méthode une des valeurs suivantes : `logging.DEBUG`, `logging.INFO`, `logging.WARNING`, etc. C'est la valeur minimal qu'ils acceptent.

Ensuite, à chaque écriture de message dans le logger, vous passez le niveau avec le message:

```
>>> logger.log(logging.DEBUG, 'Message !')
```

Comme c'est relou à taper, Python fournit des raccourcis :

`logger.debug('message')` va faire `logger.log(logging.DEBUG, 'message')`

`logger.error('message')` va faire `logger.log(logging.ERROR, 'message')`

Etc

A chaque fois que vous allez envoyer un message, le logger (et chaque handler) va comparer le niveau du message avec le sien. Si le niveau du message est inférieur au sien, il l'ignore, sinon il l'écrit.

Exemple :

```
CRITICAL
ERROR
WARNING <- niveau du logger (ou du handler)
INFO    <- niveau du message
DEBUG
```

Le message est ignoré.

```
CRITICAL
ERROR   <- niveau du message
WARNING <- niveau du logger (ou du handler)
INFO
DEBUG
```

Le message est écrit.

```
CRITICAL
ERROR   <- niveau du message et du logger
WARNING
```

```
INFO
DEBUG
```

Le message est écrit.

Ce mécanisme permet de controler la sortie de votre programme. Si vous passez le programme en prod, vous avez sans doute plus envie de tous les messages d'information et de debug. Vous mettez alors le level sur, par exemple, ERROR, et vous aurez un log avec uniquement les erreurs.

Cette techique permet un gros niveau de granularité puisque non seulement le logger à un level, mais aussi chaque handler.

Exemple : vous avez un logger avec deux handlers, un qui va vers la console, et l'autre dans un fichier.

```
CRITICAL
ERROR
WARNING  <- niveau du handler fichier
INFO    <- niveau du logger et du handler console
DEBUG
```

Si un message arrive avec un niveau DEBUG, il est ignoré. Si il arrive avec un niveau INFO, il est affiché sur la console. Si il arrive avec un niveau WARNING ou plus, il est affiché sur la console ET mis dans le fichier.

Des combos de ouf

Rappelez-vous qu'on peut avoir autant de handlers qu'on veut, et même plusieurs handlers de même nature tels que deux handlers fichiers. Le premier peut par exemple logger les infos, et l'autre juste les erreurs.

Vous pouvez même créez vos propres niveaux (`logging.CAMERDEGRAVE`) et vos propres handlers. Un handler qui appelle une URL quand il y a une erreur, un handler qui met tout le debug en coloration syntaxique, un handler qui envoit un SMS en cas de critical, etc.

Les handlers disponibles dans la lib standard sont :

- **FileHandler**: écrit tout dans un fichier.
- **RotatingFileHandler**: écrit tout dans un fichier. Si le fichier depasse une certaine taile, renomme le fichier avec un compter, et recommence avec un nouveau fichier. Le nombre de backup est configurable.
- **TimedRotatingFileHandler**: écrit tout dans un fichier. Après un certain temps, renomme le fichier avec un compter, et recommence avec un nouveau fichier. Le nombre de backup est configurable.
- **WatchedFileHandler**: écrit dans un fichier, mais surveille si il n'a pas été modifié entre temps. Ne marche pas sous Windows
- **HTTPHandler**: envoit le contenu par POST ou GET à l'URL donnée.
- **SMTPHandler**: envoit le contenu par email.
- **MemoryHandler**: garde tout en mémoire jusqu'à un certain temps, puis balance tout à un autre handler. Utile avec SMTPHandler.
- **BufferingHandler**: accumule tout en mémoire, et vide régulièrement le buffer. Utile pour l'instrospection.
- **NTEventLogHandler**: envoit les entrées sur le event log de Windows NT.
- **SysLogHandler**: envoit les entrées sur un serveur syslog.
- **SocketHandler**: envoit les entrées (picklées) via une socket.
- **DatagramHandler**: envoit les entrées sur une socket datagram.

Traiter le fichier de log

90% du temps, vous voudrez juste un bon vieux fichier de log. Mais rien ne sert d'avoir un fichier de log si on ne peut pas le lire.

D'abord, choisissez le bon endroit pour le mettre. Si c'est juste du debug, le dossier courant fera l'affaire. Si vous pensez qu'il va rester longtemps mais qu'il faudra le vider de temps à autre (et qu'il ne contient pas des informations qui ne doivent être lues que par root), alors mettez le dans un fichier temporaire. Le module tempfile est votre ami.

Sinon, pour les logs plus sérieux, il faudra le mettre dans le dossier officiel pour les logs du systèmes. Par exemple pour Ubuntu server : */var/log*.

Attention aux permissions d'écriture, vous risquez de vous retrouver avec une erreur à la con sur les bras. Vérifiez bien vos droits d'accès.

Ensuite, il faut bien choisir son format de log. Sauf cas particulier, j'utilise ce formatteur :

```
formatter = logging.Formatter('%(asctime)s :: %(levelname)s :: %(message)s'
```

Il produit des sorties du genre "TIMESTAMP :: LEVEL :: Message", par exemple :

```
2012-03-08 12:01:31,311 :: INFO :: My life for Aiur
2012-03-08 12:20:31,311 :: ERROR :: We require more Vespin Gaz
2012-03-08 12:37:31,311 :: WARNING :: We are under attack
2012-03-08 12:38:31,311 :: CRITICAL :: Nuclear launch detected
```

Ce format est facile à analyser car le séparateur a peu de risque de se retrouver dans le message.

On peut très rapidement filtrer uniquement les erreurs, par exemple sous bash on peut voir si il y a eu récement des erreurs ainsi :

```
tail -n 100 fichier.log | grep ERROR
```

Sous Python on peut faire des traitements super chiadés en deux secondes :

```
from datetime import datetime

lines = (ligne.split(' :: ') for ligne in open('fichier.log'))
errors = ((date, mes) for date, lvl, mes in lines if lvl in ('ERROR', 'CRIT

before, after = datetime(2012, 1, 12), datetime(2012, 3, 24)
parse = lambda d: datetime.strptime(d, '%Y-%m-%d %H:%M:%S,%f')
dated_line = ((date, mes) for date, mess in errors if before <= parse(date)

for date, message in dated_line:
    print date, message

# Affiche uniquement les message d'erreur ou criques qui sont arrivés entre
#  le 12 janviers 2012 et le 24 mars 2012
```

Et je ne vous parle même pas de ce qu'on peut faire avec des libs comme pandas.

Aller plus loin

Le log peut vous emener très loin. En vérité, non seulement on peut avoir plusieurs handlers, mais on peut aussi avoir plusieurs loggers.

Quand vous faites:

```
logger = logging.getLogger()
```

Vous récupérez ce qu'on appelle le logger "racine" (ou "root"). Mais vous pouvez aussi donner un nom au logger :

```
logger = logging.getLogger('encoding')
```

Ceci vous permet de préconfigurer le logger quelque part dans le programme, et de récupérer celui-ci en particulier à un autre endroit, sans se trimbaler la référence. `logging.getLogger('encoding')` n'a besoin d'être configuré qu'une fois, si vous refaites ça ailleurs, vous récupérez le même logger.

On peut même utiliser des sous-noms:

```
enc_aud_logger = logging.getLogger('encoding.audio')
enc_vid_logger = logging.getLogger('encoding.video')
```

Alors chaque logger aura sa propre config, MAIS, tout message envoyé sur un logger nommé "encoding.quelquechose" sera aussi automatiquement envoyé au logger "encoding" si il existe.

Vous pouvez donc créer une hierarchie de log, par exemple pour avoir un fichier par process, et un gros log central sur un serveur à part via socket. Ca ne m'a jamais servi, mais je me suis dit que je ferai circuler l'info.

Je n'ai pas parlé des filters, qui sont des objets qui permettent de faire la même chose que les niveaux, mais avec des règles d'écriture dans les logs plus avancées que juste "c'est le bon niveau". Et je n'ai pas abordé non plus les formateurs particuliers. Là on rentre dans le domaine de la drosophilie.

En revanche si vous avez le temps, je vous conseille de passer quelque temps sur les options de configuration (plein de recettes ici). En effet, en général on veut avoir un log configurable (c'est un peu le but de la manœuvre, sinon on ferait des print). Or, le logger peut se configurer de plein d'autres façons qu'avec du code Python.

On peut notamment déclarer la configuration avec un gros dictionaire (c'est ce que fait Django) ou avec un fichier ini.

Enfin, normalement ce package est thread-safe. Vous devriez pouvoir logger en toute impunité dans plusieurs threads. Je recommande quand même de faire la configuration du logger avant d'entrer dans un thread, on ne sait jamais.

Pour ceux qui veulent se plonger dans les méandres de tout ce qu'on peut faire avec logging, chechez la doc et cet article de l'auteur du